Ahmet Emre Tümer
152120171021

Introduction to Artificial Intelligence Lab

Lab 3: Uninformed Search Strategies

Uninformed strategies use only the information available in the problem definition. Uninformed Search includes the following algorithms:

- Breadth First Search (BFS)
- Depth First Search (DFS)
- Uniform Cost Search (UCS)
- Depth Limited Search (DLS)
- Iterative Deepening Search (IDS)
- Bidirectional Search (BS)

BFS

- Expand shallowest unexpanded node.
- In breadth-first search the frontier is implemented as a **FIFO** (first-in, first-out) queue. Thus, the path that is selected from the frontier is the one that was added earliest.
- This approach implies that the paths from the start node are generated in order of the number of arcs in the path. One of the paths with the fewest arcs is selected at each stage.

Pseudocode:

Input: A graph G and a starting vertex v of G

Output: All vertices reachable from v labeled as explored.

A non-recursive implementation of breadth-first search:

```
1 Breadth-First-Search(G, v):
2
3     for each node n in G:
4         n.distance = INFINITY
5         n.parent = NIL
6
7     create empty queue Q
8
9     v.distance = 0
10    Q.enqueue(v)
11
12    while Q is not empty:
13
14        u = Q.dequeue()
15
16        for each node n that is adjacent to u:
17            if n.distance == INFINITY:
18                n.distance = u.distance + 1
19                n.parent = u
20                Q.enqueue(n)
```

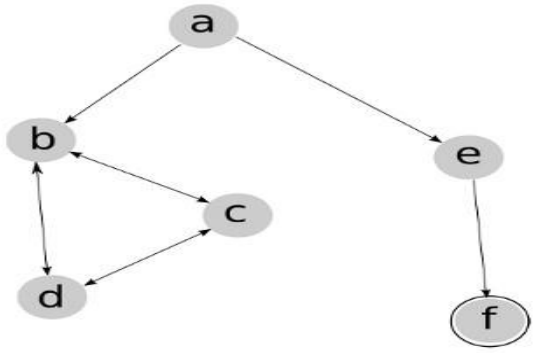


Fig 1.

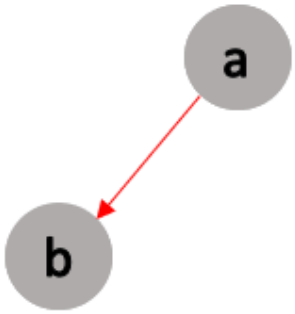
Exercise 1:

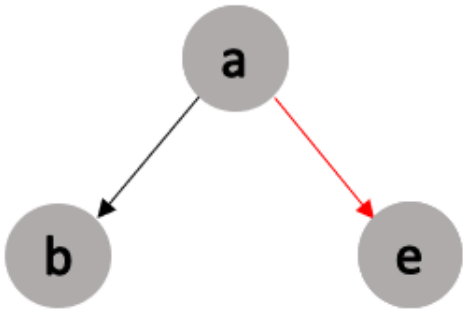
- Write a program to implement BFS in Fig 1.
- Show which sequences of paths are explored by BFS.
- Use queue/stack data structure that you implemented in Experiment 2.

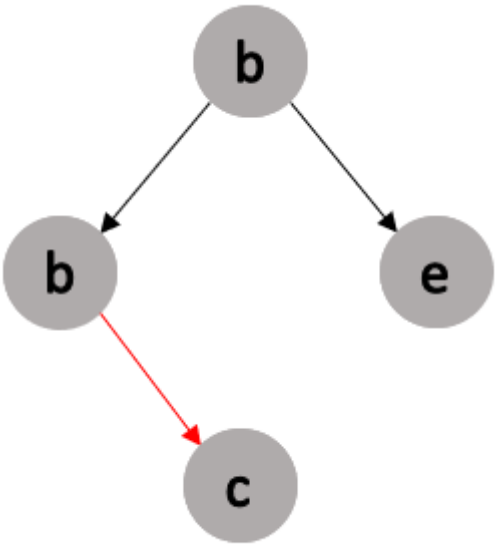
```cpp
#include <stdio.h>
#include <iostream>
#include <queue>
#include <list>
using namespace std;

void printit(int no) {
	if (no == 0)
	{
		cout << "a" << " ";
	}
	else if (no == 1)
	{
		cout << "b" << " ";
	}
	else if (no == 2)
	{
		cout << "c" << " ";
	}
	else if (no == 3)
	{
		cout << "d" << " ";
	}
	else if (no == 4)
	{
		cout << "e" << " ";
	}
	else if (no == 5)
	{
		cout << "f" << " ";
	}
}

class Graph
{
	int nodeS; // Node sayisi
	vector<list<int>> adj;

public:

	Graph(int V); // Constructor fonksiyon

	void addPath(int node, int target); // Nodeların yollarını tanımlayan fonksiyon

	void BreadthFirstSearch(int no); // Breadth First Search algoritmasının izlediği
yolu gösterir.
};

Graph::Graph(int V)
{
	this->nodeS = V;
	adj.resize(V);
}
```

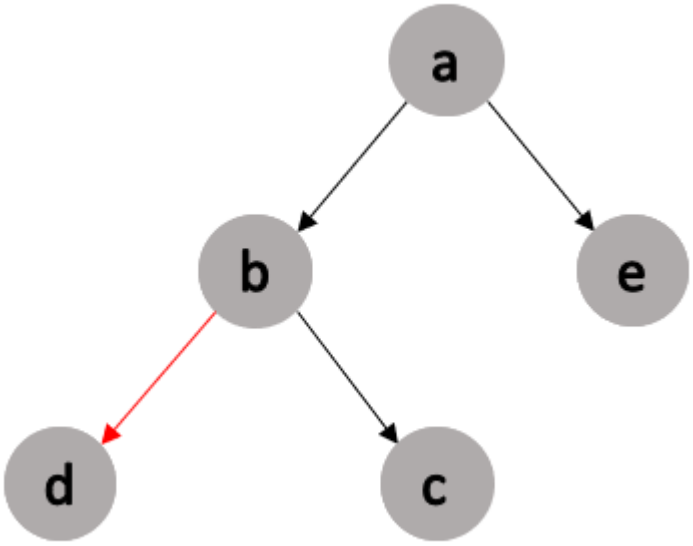
```
void Graph::addPath(int node, int target)       // Nodeların yollarını tanımlayan
fonksiyon
{
	adj[node].push_back(target); //
}

void Graph::BreadthFirstSearch(int no)
{
	// Nodeları gezilmemiş olarak işaretler
	vector<bool> visited;
	visited.resize(nodeS, false);

	// Breadth First Search için queue oluşturma
	list<int> queue;

	// Üstünde olduğumuz nodeu gezilmiş olarak işaretle ve enqueue et.
	visited[no] = true;
	queue.push_back(no);

	while (!queue.empty())
	{
		no = queue.front();
		printit(no);  // Yazdırma fonksiyonu
		queue.pop_front();

		// Dequeue edilen tüm komşu nodeları al
		// eğer komşu node ziyaret edilmemişse
		// ziyaret edildi olarak işaretle ve enqueue işlemini yap
		for (auto adjacent : adj[no])
		{
			if (!visited[adjacent])
			{
				visited[adjacent] = true;
				queue.push_back(adjacent);
			}
		}
	}
}

int main()
{
	// 7 nodelu bir graph oluşturma
	Graph g(7);

	// Nodeların diğer nodelara olan yollarını tanımlama
	g.addPath(0, 1);    // a -> b
	g.addPath(0, 4);    // a -> e
	g.addPath(1, 2);    // b -> c
	g.addPath(1, 3);    // b -> d
	g.addPath(2, 1);    // c -> b
	g.addPath(2, 3);    // c -> d
	g.addPath(3, 1);    // d -> b
```

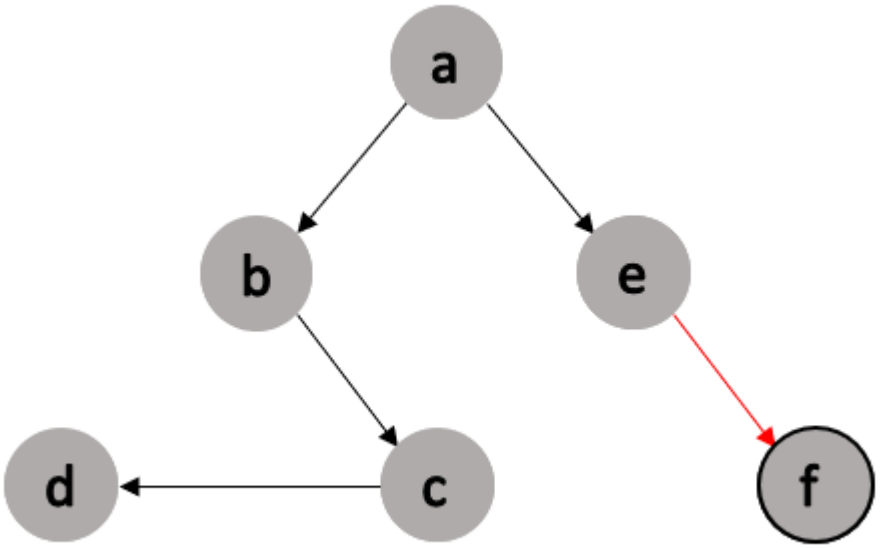
```
        g.addPath(3, 2);     // d -> c
        g.addPath(4, 5);     // e -> f
        g.addPath(5, 5);     // f -> f

        // Breadth First Search sonuçları
        g.BreadthFirstSearch(0);

        return 0;
}
```

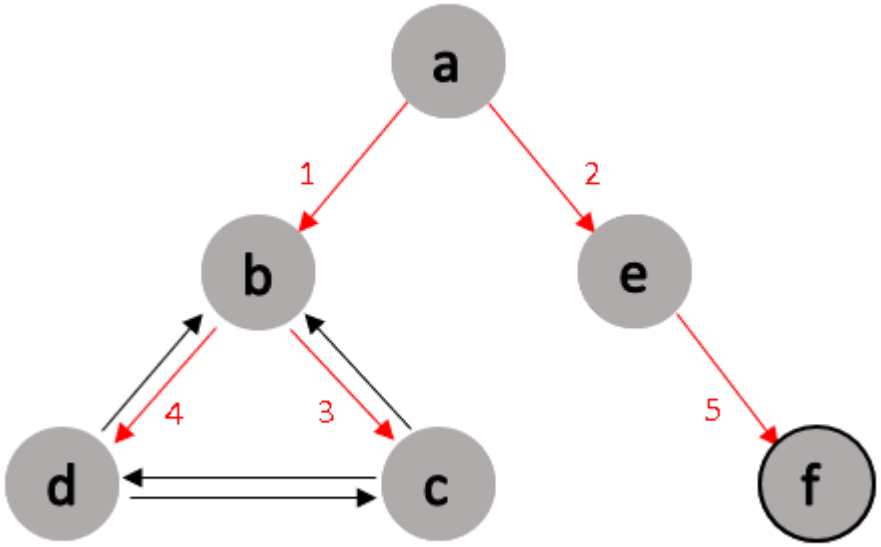


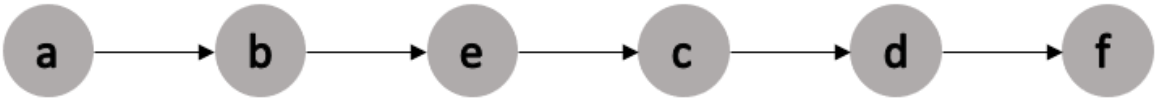







a
b
e
d
c
f


a
1
2
b
e
4
3
5
d
c
f


a
b
e
c
d
f